

21190062 – Avelino José Correia Martins

21180276 – Filipe Rafael Lopes Brandão

***Implementação de um clone do jogo Arkanoid em Java***

Trabalho desenvolvido no âmbito da disciplina Programação Avançada, do 2º ano da Licenciatura em Engenharia Informática.

Coimbra

Junho, 2011

**Índice**

Conteúdo

[INTRODUÇÃO 2](#_Toc295672054)

[CAPÍTULO 1 – O MOTOR DE JOGO 3](#_Toc295672055)

[CAPÍTULO 2 – AS CLASSES BASE 4](#_Toc295672056)

[BOLA 4](#_Toc295672057)

[RAQUETE 4](#_Toc295672058)

[TIJOLO 5](#_Toc295672059)

[BÓNUS 5](#_Toc295672060)

[JOGADOR 6](#_Toc295672061)

[NÍVEL DE JOGO 6](#_Toc295672062)

[PAREDES 6](#_Toc295672063)

[CAPÍTULO 3 – AREA DE JOGO 7](#_Toc295672064)

[CAPÍTULO 4 – PADRÕES DE DESENHO 8](#_Toc295672065)

[PADRÃO MVC 8](#_Toc295672066)

[PADRÃO FACTORY 8](#_Toc295672067)

[PADRÃO COMANDO 9](#_Toc295672068)

[PADRÃO MÁQUINA DE ESTADOS 9](#_Toc295672069)

[PADRÃO SINGLETON 10](#_Toc295672070)

[CAPÍTULO 5 – MENUS E CAIXAS DE DIÁLOGO 11](#_Toc295672071)

[MENU INICIAL 11](#_Toc295672072)

[MENU PAUSA 12](#_Toc295672073)

[CAIXA DE DIÁLOGO LOGIN 12](#_Toc295672074)

[CAIXA DE DIÁLOGO REGISTO 13](#_Toc295672075)

[CAIXA DE DIÁLOGO REINICIAR RÁPIDO 13](#_Toc295672076)

[CAIXA DE DIÁLOGO TOP10 14](#_Toc295672077)

[CAPÍTULO 6 – ECRÃS 15](#_Toc295672078)

[CAPÍTULO 8 – ITENS PERSONALIZADOS 17](#_Toc295672079)

[CONCLUSÃO 18](#_Toc295672080)

# INTRODUÇÃO

A linguagem de programação Java, caracterizada especialmente pela sua portabilidade e capacidade de gestão de memória automaticamente, é uma linguagem orientada a objectos influenciada por C, C++, entre outras. Este tipo de programação é uma em que os programadores não definem apenas o tipo de uma estrutura de dados mas também o tipo de operações (funções) que podem ser aplicados a esses mesmos dados. Desta forma a estrutura de dados passa a ser um *objecto*. Em adição os programadores podem criar relações entre objectos. Por exemplo os objectos podem herdar características de outros objectos.

Uma das principais vantagens de técnicas de programação orientada a objectos sobre técnicas de programação procedimental, é que elas permitem aos programadores criar módulos (classes) que não necessitam de serem alterados quando um novo tipo de objecto é adicionado. Um programador pode simplesmente criar um novo objecto que herde muitos dos recursos de objectos já existentes.

Como um objecto (classe) define apenas os dados com que necessita de se preocupar ou que o descrevem, quando uma instância desta classe é executada, o código não será capaz de aceder acidentalmente a outros dados do programa. Esta característica de ocultação de dados proporciona uma maior segurança do sistema e evita corrupção de dados não desejada.

Estas e outras características serão sem dúvida de extrema utilidade para resolver questões incontornáveis que surgem imediatamente após uma análise do enunciado do projecto, e que deverão ser tidas em conta para um planeamento correcto que permita uma implementação robusta, flexível, de fácil manutenção e também expansão.

Graças à capacidade de gestão de memória automaticamente, o programador consegue abstrair-se e programar toda aplicação sem se preocupar com a libertação de recursos. Isto porque o mecanismo *Garbage Collector*,proporcionado pelo próprio ambiente de execução JVM, fica encarregue de gerir estes aspectos.

Anexo a este relatório, encontram-se um diagrama de pacotes e vários diagramas de classes resultantes da implementação desta apliação.

# CAPÍTULO 1 – O MOTOR DE JOGO

Desde o iníco deste Trabalho Prático, decidimos por iniciativa própria usar de alguma forma a API OpenGL no desenvolvimento da interface gráfica. Foi então que após alguma pesquisa descobrimos a biblioteca Lightweight Java Game Library.

Esta biblioteca permite, de forma simples, o acesso às potencialidades do OpenGL. Assim, é possível desenvolver aplicações altamente dinâmicas (característica que pretendíamos incluir no nosso trabalho).

De forma a aproveitar as potencialidades desta biblioteca, foi desenvolvida uma classe denominada ***GameEngine***. Esta classe é composta essencialmente por atributos que definem como é feita a análise de inputs (***mCurrentController***) e como é definida a interface do jogo (***mCurrentView***). Além destes atributos, existem algumas funções fundamentais que proporcionam a base de funcionamento de toda a aplicação. Essas funções são:

* **setup** : cria o Display, Keyboard e Mouse de acordo com as configurações pretendidas;
* **initGL** : define as configurações OpenGL;
* **run** : define a inicialização da aplicação e o ciclo de actualização contínua. É aqui que é despoletada a actualização da interface e a análise de inputs.

Outra função desta classe que pode ser considerada importante é a ***checkGameState***. Esta função define o comportamento a tomar de acordo com o estado em que o jogo se encontra. O seu funcionamento será revisto no capítulo “Padrões de Desenho” durante a explicação da implementação da Máquina de Estados.

# CAPÍTULO 2 – AS CLASSES BASE

“Os elementos principais deste jogo são a bola, a raquete, tijolos coloridos e os bónus que surgem quando alguns tijolos são destruídos.”

Através da interpretação do enunciado do Trabalho Prático, foi possível definir um conjunto de classes necessárias ao funcionamento básico desta aplicação. Antes de mais, é importante referir que, dado que grande parte das classes base dependem da capacidade de detecção de colisões (exemplo: bola colide com raquete), foi necessário desenvolver classes que definem as funcionalidades imprescindíveis a estes comportamentos. Estas classes foram denominadas ***BoundingBox*** (que define a caixa delimitadora de um objecto gráfico) e ***Collidable*** (que é composta por ***BoundingBox***’s e implementa realmente as funções de detecção de colisões).

## BOLA

Dado o facto de a bola ser uma entidade composta que tem de estar em constante movimento (alterável por entidades exteriores), foi definida a classe ***Ball*** que encapsula o seu funcionamento essencial. Esta classe é composta fundamentalmente por um vector de deslocamento, reflectido em dois atributos que definem o deslocamento no eixo X e no eixo Y. Além disto, possui ainda algumas flags de controlo que surgiram da necessidade de implementar funcionalidades de outras entidades que interagem com esta. Tal como por exemplo o atributo ***mIsGluedToClub*** que é necessário para implementar o funcionamento do Bónus Cola (descrito mais à frente) e do início do jogo (em que a bola começa colada à raquete).

As funções implementadas nesta classe, são basicamente *getters* e *setters,* à excepção da função ***updatePosition***que é usada para que entidades exteriores (que encapsulem esta) possam actualizar a posição da bola a qualquer momento (e, por conseguinte, da sua ***BoundingBox***). A suportar o funcionamento desta função, existe um atributo que indica o tempo em que a posição da bola foi alterada pela última vez, usado para descobrir o tempo que passou entre essa alteração e o momento em que a função é invocada. Desta forma é possível manter a sincronização independente do computador em que a aplicação é executada.

## RAQUETE

A classe que representa esta entidade foi denominada ***Club***. Trata-se no fundo do principal elemento de interacção com o jogador. No entanto é de uma implementação simples visto que apenas necessita de ser posicionada consoante os inputs recebidos (que são tratados noutra classe central). Como essa função já se encontra definida na classe da qual deriva (função ***setX*** da classe ***Collidable***) a única função que os autores acharam útil implementar foi a ***placeBallAtCenter*** que, recebendo uma ***Ball***, é capaz de a posicionar no centro da raquete.

CAPÍTULO 2 – AS CLASSES BASE

## TIJOLO

A classe base dos tijolos foi definida como ***Brick*** e trata-se também de uma entidade com tratamento de colisões. Estes elementos possuem uma flag ***mIsActive*** que indica se estão activos (útil para futuramente detectar quais os tijolos a apresentar no ecrã) e, dada a necessidade do Tijolo Verde só ser destruído na 2ª colisão, foi também incluído um atributo para registar o número de colisões que um tijolo já sofreu. Esta classe possui uma função overloaded ***onBallColision*** que define o comportamento a tomar quando existe uma colisão da bola com o tijolo. Esta função, quando recebe uma ***Ball*** como argumento, altera a direcção da bola. E quando o argumento é toda a área de jogo, o comportamento a aplicar já depende do tipo de bónus em questão, sendo então redefinido nas classes derivadas. Por exemplo: o Tijolo Azul instancia um bónus na sua posição enquanto o Tijolo Vermelho destrói todos os outros que se encontram em posições adjacentes.

## BÓNUS

Estes elementos são definidos na classe ***Bonus*** e são instanciados na área de jogo pelo Tijolo Azul quando é destruído. No fundo, deslocam-se verticalmente segundo uma velocidade definida e inalterável durante o decorrer do jogo. Essa deslocação é efectuada pela função ***updatePosition***. Existem ainda as funções ***onClubCollision*** e ***undoEffect*** que invocam o efeito provocado pelo bónus quando é apanhado pelo jogador (isto é, quando colide com a raquete) e anulam esse mesmo efeito (visto que um bónus quando apanhado anula o anterior). Estas duas funções são redefinidas em cada uma das classes derivadas visto que cada tipo de bónus tem o seu efeito específico.

CAPÍTULO 2 – AS CLASSES BASE

## JOGADOR

A classe que representa um jogador no contexto do jogo propriamente dito, é composta essencialmente por atributos que indicam o número de vidas e a pontuação do jogador e respectivos *getters* e *setters*.

Existe ainda a classe ***RegisteredPlayerData*** que é usada noutro contexto: representar a informação do jogador registado (username, password, níveis completados).

## NÍVEL DE JOGO

Um nível de jogo também é representado por uma classe denominada ***GameLevel***. Os seus atributos reflectem as configurações de cada nível. Configurações essas que se tratam do seu nome, conjunto de tijolos, número de vidas inicial, entre outros. Visto que os níveis são guardados em ficheiros de texto, foram definidas funções essenciais para os carregar instanciando todos os objectos necessários à medida que o ficheiro de um nível é processado. Para tal, foi definida a função readLevelFile que, indicando como argumento o nome do ficheiro a carregar, usa as funções auxiliares ***processLevelFileLineByLine***, ***processInfoLine*** e ***processBricksLine*** para efectuar parsing dos dados e instanciar todos os objectos necessários.

## PAREDES

Foi também definida uma classe ***Wall*** para representar uma parede da área de jogo. Nesta classe são considerados 3 tipos de parede: parede esquerda, superior e direita. Foi definida a função ***doEffectOnBall*** com o intuito de a invocar quando é detectada uma colisão da bola com uma parede. Esta função apenas altera a trajectória da bola dependendo do tipo de parede. Como auxiliar para mais tarde controlar a posição da raquete, foi definida uma função ***doEffectOnClub*** que, caso exista colisão entre estas duas entidades, não permite que a raquete ultrapasse os limites.

# CAPÍTULO 3 – AREA DE JOGO

Todas as entidades referidas no capítulo anterior são importantes no contexto do jogo em si. Isto é, todas elas fazem parte da área de jogo com a qual o jogador consegue interagir.

Posto isto, foi definida a classe ***PlayArea*** que é composta por todos os elementos referidos anteriormente. Desta forma, torna-se uma entidade essencial e importantíssima nesta aplicação, já que funciona como um elo de ligação onde são feitas verificações relevantes a cada ciclo de execução do jogo. O papel central desta classe é facilmente comprovado no seguinte diagrama de classes conceptual:

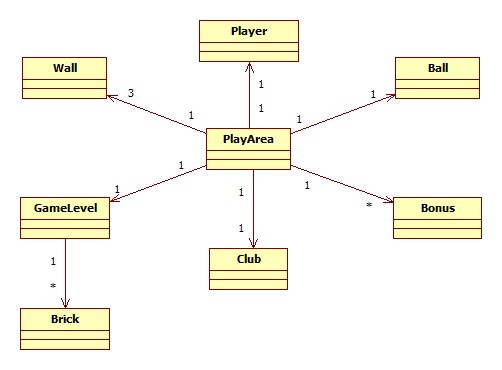


Figura 1 - Modelo Conceptual da Área de Jogo

Para implementar todas as verificações necessárias, foram definidas funções que são invocadas a cada *tick* do jogo. Essas funções servem no fundo para detectar colisões e verificar se alguns elementos atingiram o limite inferior da área de jogo. Como exemplo, existe a função ***checkBallBrickCollisions*** que percorre os tijolos do ***GameLevel***, de forma a verificar se a bola está a colidir com algum deles. Resumidamente, caso se verifique essa colisão, é invocada a função ***onBallCollision*** do tijolo e, se necessário, é incrementado o número de tijolos destruídos.

Todas as funções de verificação são chamadas no método ***tick*** que é invocado de forma contínua a partir do motor de jogo (classe ***GameEngine***).

# CAPÍTULO 4 – PADRÕES DE DESENHO

Os padrões de desenho são soluções que se adequam a problemas de desenho recorrentes e simplificam o processo de análise e desenvolvimento de Software. No entanto, é preciso ter em conta que estes padrões não oferecem “soluções enlatadas” para todos os elementos. Isto implica que não exista uma implementação específica dos mesmos, mas sim uma adaptação às necessidades concluídas depois de analisar bem os problemas.

Neste capítulo, serão descritos os padrões de desenho implementados no Trabalho Prático.

## PADRÃO MVC

Consultar diagrama de classes na página 11 do Anexo.

Desde o início, existiu especial atenção a implementar este padrão de arquitectura, adaptando-o às bibliotecas que estamos a usar. Não só por ser considerado obrigatório no enunciado, mas também porque temos a completa noção das suas vantagens e da importância deste padrão nas aplicações modernas. O melhor caso da sua aplicação no nosso trabalho reflecte-se nas classes: ***ViewPlayAreaGeom*** (Vista), ***ModelPlayArea*** (Modelo) e ***GameAreaController*** (Controlador).

A classe ***GameAreaController*** encontra-se directamente ligada ao motor de jogo ( ***GameEngine*** ) a escutar o input (rato e teclado) durante o decorrer do jogo, fazendo o papel de Controlador. A todo o momento, a informação captada é redireccionada para o Modelo que encapsula apenas as funcionalidades necessárias da classe ***PlayArea*** (composta por todos os dados do jogo em si). Por outro lado, existe a classe ***ViewPlayAreaGeom*** também ligada ao motor de jogo, que se encontra a desenhar todos os elementos necessários no ecrã durante todo o desenrolar do jogo. Dado que esta se trata de uma aplicação bastante dinâmica e o próprio motor de jogo controla o momento em que o jogo é redesenhado, não existiu necessidade de implementar o padrão Observador que foi leccionado durante as aulas.

## PADRÃO FACTORY

Consultar diagrama de classes na página 9 do Anexo.

Existem vários objectos que são instanciados dependendo de várias condições que só se verificam em runtime. Um exemplo deste funcionamento está na instanciação dos tijolos durante o início de cada nível. Isto é, os tijolos a instanciar no ***GameLevel***, depende intrinsecamente do nível que é carregado. Por esta razão, à medida que o ficheiro de nível é carregado, são instanciados os tijolos necessários por intermédio da classe ***BrickUtils*** (a dita Factory) que possui uma função *static* com o objectivo de instanciar e retornar um tijolo dependendo das características que lhe são passadas por argumento.

CAPÍTULO 4 – PADRÕES DE DESENHO

## PADRÃO COMANDO

Consultar diagrama de classes na página 12 do Anexo.

Este padrão foi útil na implementação da funcionalidade de Replay. Isto porque existiram vários eventos que necessitaram de ser armazenados sequencialmente para futura reprodução. Esses eventos são por exemplo: a passagem de nível (classe **ChangeLevelEvent**), uma tecla pressionada (**KeyPressEvent**), movimento e clique do rato (**MouseEvent**) e actualização temporal da área de jogo (**TickEvent**). Os eventos atrás mencionados, foram usados com base no facto de que o jogo na sua maioria é determinístico. Existe a questão do evento não determinístico que é a criação de bónus por parte dos tijolos. Estes bónus são criados de forma aleatória. Assim sendo, usámos também um evento (**ChangeBonusSeedEvent**) para guardar a *seed* usada para inicializar a classe de sistema Random. Usando o padrão Comando na implementação desta funcionalidade, foi possível encapsular os eventos que constituem um Replay, para mais tarde os executar ou navegar entre eles como se uma linha de tempo se tratasse.

## PADRÃO MÁQUINA DE ESTADOS

Consultar diagrama de classes na página 10 do Anexo.

Decidimos aplicar este padrão dado que, nesta aplicação, podem ser considerados vários estados que variam em runtime. Daí surge um comportamento diferente da aplicação dependendo do estado em que se encontra. Esta informação é útil ao motor de jogo que se comportará tendo em conta esse mesmo estado em que se encontra.

Para implementar este funcionamento, recorremos a uma classe ***GameState*** onde estão definidos todos os estados possíveis e as funções que permitem alterar o estado actual de forma simples. Como exemplo, temos definidos os estados “MAIN\_MENU”,”PLAYING” e ”GAME\_OVER”, entre outros. Quando a aplicação é iniciada, encontra-se no estado “MAIN\_MENU” que indica que deve ser apresentado o menu inicial. Assim que o utilizador inicia o jogo, o estado passa a “PLAYING” e assim o motor de jogo reconhece que deve iniciar o seu funcionamento propriamente dito (desenhar os elementos no ecrã e actualizar de forma contínua de acordo com os inputs do jogador). Se o jogador perder todas as vidas, o estado é automaticamente passado para “GAME\_OVER” e assim o motor de jogo limita-se a apresentar o menu de game over.

CAPÍTULO 4 – PADRÕES DE DESENHO

## PADRÃO SINGLETON

Consultar diagrama de classes nas páginas 2 a 6 do Anexo.

Nesta aplicação existem várias classes das quais só é necessário um objecto durante toda a execução, optámos por criar os singletons que nos fornecerão um acesso global e, ao mesmo tempo, tornarão a comunicação entre os objectos mais simples e intuitiva. Um bom exemplo disso é a classe ***GameState*** referida anteriormente. Apesar de não ter sido leccionado, achámos que a implementação deste padrão seria apropriada e útil.

Como já foi referido, aplicámos este padrão na classe ***GameState***, visto que só é instanciado um único objecto e é necessário o acesso global a ele, simplificando alguns pormenores de implementação. Outras classes onde implementámos este padrão foram as ***RegisteredPlayerData*** (que contém os dados dos utilizador autenticado), ***Top10*** (que contém os dados a apresentar na caixa de diálogo do Top10) e ***Textures*** e ***Sounds*** (que carregam apenas uma vez todos os recursos multimédia necessários).

Todas as classes onde é implementado este padrão, possuem um construtor privado para que seja impedida a instanciação noutras classes. Sempre que se quer aceder a um dos singletons, usa-se o nome da classe e o método *getInstance*.

# CAPÍTULO 5 – MENUS E CAIXAS DE DIÁLOGO

De forma a aplicar os conceitos leccionados acerca do desenvolvimento de GUI’s com o Swing, foram implementados alguns menus e caixas de diálogo recorrendo a esse toolkit.

## MENU INICIAL

Menu apresentado no início da aplicação. O seu conteúdo varia caso o jogador esteja autenticado ou não.

|  |  |
| --- | --- |
| C:\Users\Filipe\Desktop\1.png  Figura 2 - Menu Inicial sem autenticação | C:\Users\Filipe\Desktop\2.png  Figura 3 - Menu inicial com autenticação |

CAPÍTULO 5 – MENUS E CAIXAS DE DIÁLOGO

## MENU PAUSA

Menu apresentado quando é pressionada a tecla Escape durante o jogo ou quando o jogador perder.

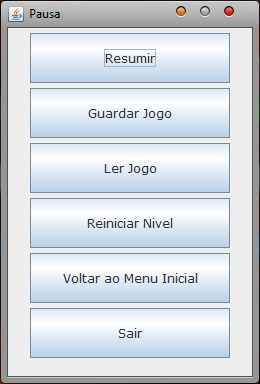


Figura 4 - Menu Pausa

## CAIXA DE DIÁLOGO LOGIN

Apresentado quando é pressionado o botão de Login no Menu Inicial e permite ao utilizador autenticar-se perante a aplicação.

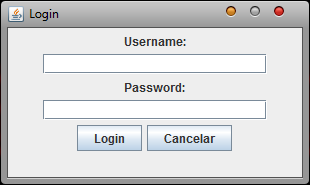


Figura 5 - Caixa de Diálogo de Login

CAPÍTULO 5 – MENUS E CAIXAS DE DIÁLOGO

## CAIXA DE DIÁLOGO REGISTO

Apresentado quando é pressionado o botão de Registo no Menu Inicial e permite ao utilizador registar-se na aplicação para que mais tarde se possa autenticar.

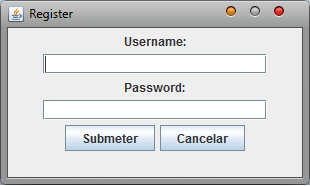


Figura 6 - Caixa de Diálogo de Registo

## CAIXA DE DIÁLOGO REINICIAR RÁPIDO

Apresentado quando é pressionado o botão de Reiniciar Rápido no menu inicial, apenas quando o utilizador está autenticado e já completou no mínimo o primeiro nível. Permite ao utilizador, começar o jogo num nível que já tenha atingido anteriormente.

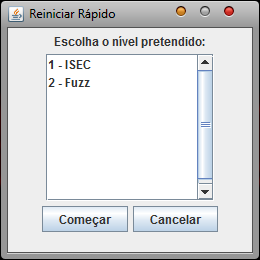


Figura 7 - Caixa de Diálogo de Reinicio Rápido

CAPÍTULO 5 – MENUS E CAIXAS DE DIÁLOGO

## CAIXA DE DIÁLOGO TOP10

Apresentado quando é pressionado o botão de Top10 no menu inicial. Permite verificar a tabela de pontuações e seguir para o Replay de um desses jogos (caso o utilizador esteja autenticado).

|  |  |
| --- | --- |
| C:\Users\Filipe\Desktop\6.png  Figura 8 - Tabela de Top10 | C:\Users\Filipe\Desktop\7.png  Figura 9 - Tabela de Top10 para utilizador autenticado |

# CAPÍTULO 6 – ECRÃS

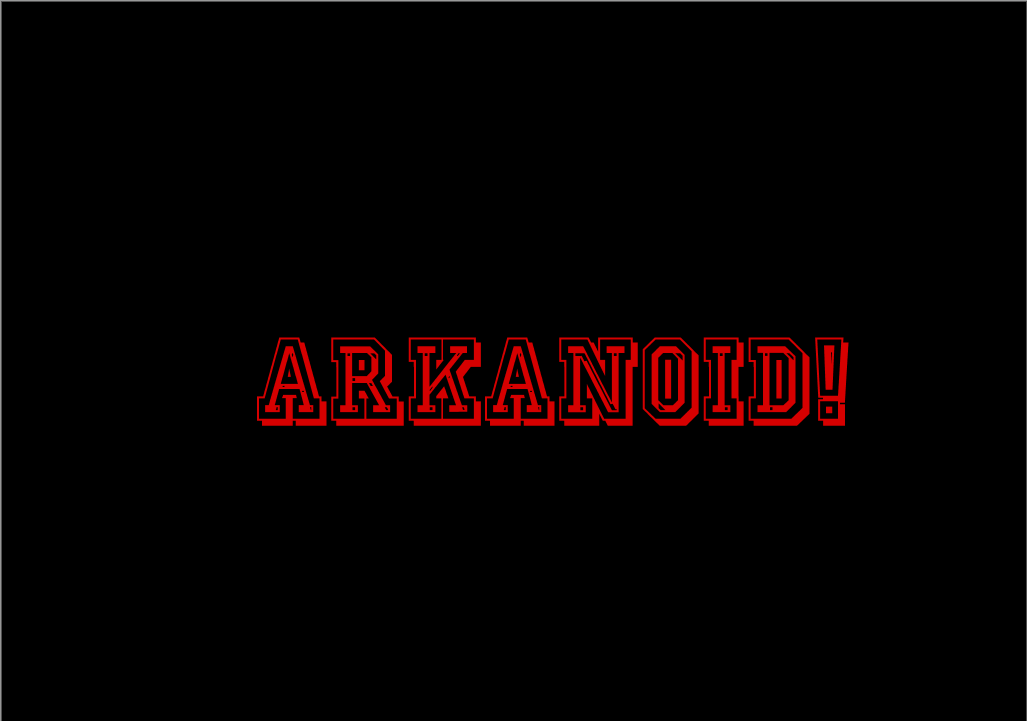


Figura 10 - Ecrã Inicial

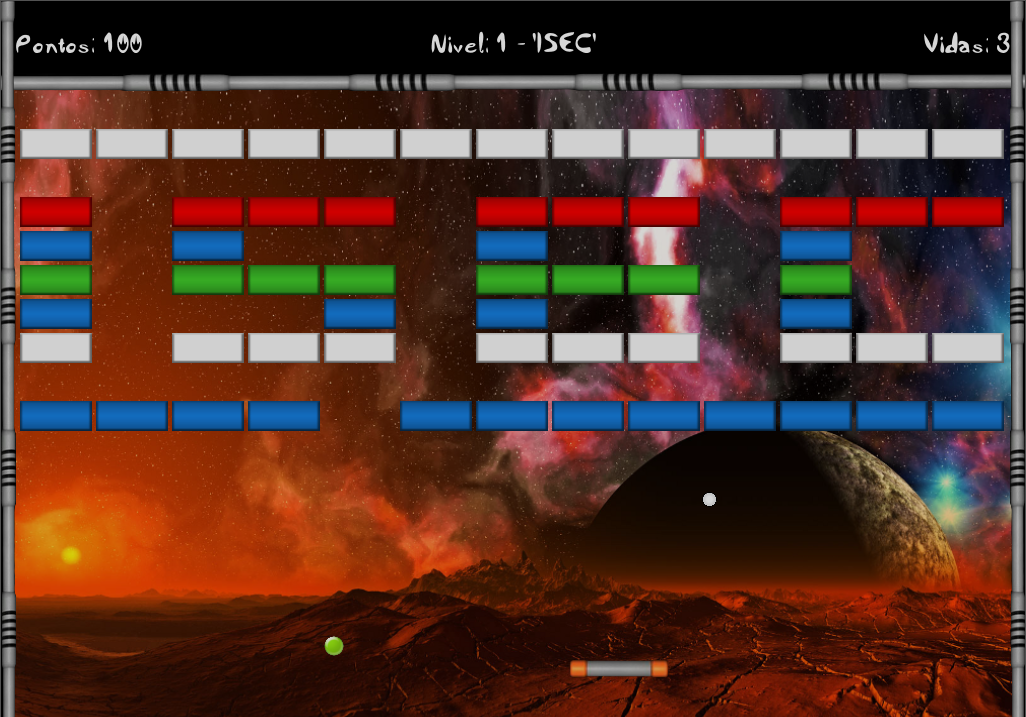


Figura 11 - Ecrã de Jogo com texturas

CAPÍTULO 7 – ECRÃS

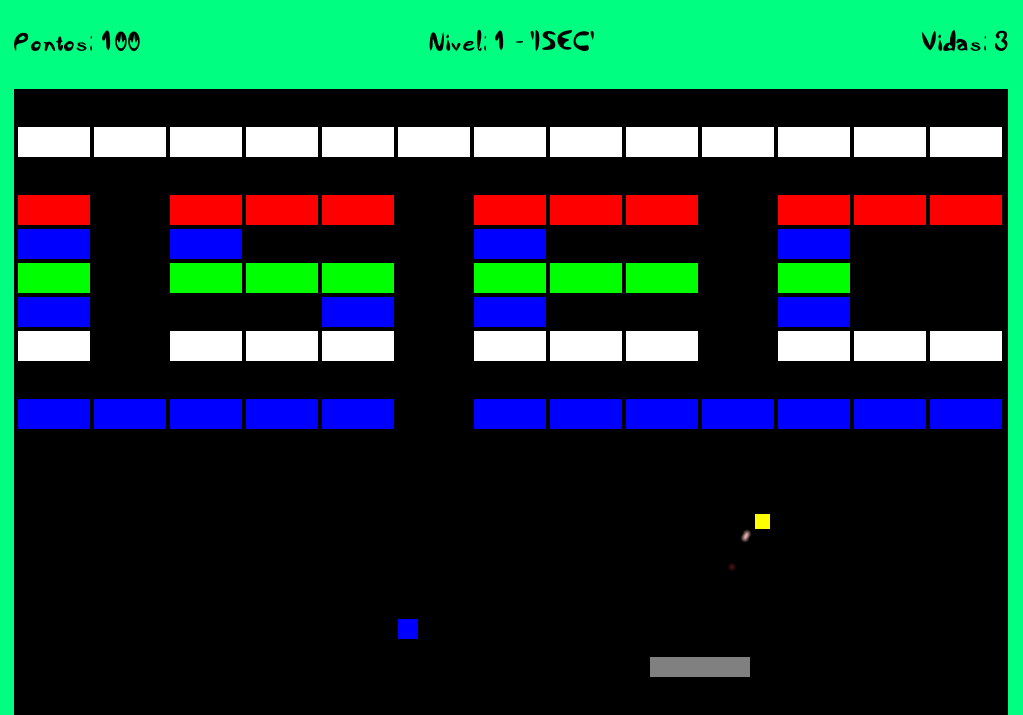


Figura 12 - Ecrã de Jogo sem texturas



Figura 13 - Ecrã de Game Over

# CAPÍTULO 8 – ITENS PERSONALIZADOS

Aqui apresentamos as nossas escolhas relativamente aos itens personalizados que eram pretendidos. Tivemos várias ideias, mas optamos por soluções simples em que não foi necessário andar a alterar o código de outras classes apenas por causa destas novas entidades.

**TIJOLO**

O tijolo personalizado criado foi o tijolo Invisível que, como o nome indica, não é visível na área de jogo apesar de existir e ser necessário destruí-lo para terminar um nível.

**BÓNUS**

O bónus personalizado criado foi o bónus Inverte. Este bónus na realidade é uma armadilha já que troca o input do jogador. Isto é, quando o jogador apanha este bónus, a tecla da Direccional Esquerda move a raquete para a direita e a Direccional Direita move a raquete para a esquerda. O mesmo acontece com o rato.

# CONCLUSÃO

Foram adquiridas com a execução deste projecto competências no que diz respeito a programação orientada a objectos em Java assim como na metodologia em geral. Acreditamos que, a aprendizagem desta linguagem de programação, será bastante útil para o nosso futuro. Principalmente se for considerado que é uma das linguagens mais populares e existe uma grande probabilidade de nos voltarmos a cruzar com ela no “mundo do trabalho”.

Visto que no semestre passado já tínhamos frequentado as aulas da cadeira de Programação Orientada a Objectos, mais uma vez verificou-se que é extremamente rápida a velocidade com que se percebem as vantagens obtidas pelo uso de uma metodologia POO relativamente a uma metodologia procedimental em projectos deste tipo.

As questões que apareceram durante a implementação, foram também importantes no aspecto que, ao pensar sobre elas e a forma de as contornar, levou-nos a compreender a importância dos conceitos aprendidos nas aulas especialmente o uso dos padrões de desenho.

Como conclusão, consideramos que foram atingidos todos os objectivos pretendidos neste Trabalho Prático. Julgamos também que o facto do enunciado proposto ser bastante interessante e dinâmico tornou divertida a sua implementação. O único ponto menos positivo acerca deste trabalho prático, foi a questão da funcionalidade de Replay que, depois de imensas horas perdidas, nos pareceu demasiado complexa para ser aplicada de forma totalmente correcta. Ainda que não sendo 100% correcta, consideramos que a nossa implementação desta funcionalidade vai de encontro às expectativas.